VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vyhledávání Vzorů v Komprimovaných Datech

Compressed Pattern Matching

2019 Michal Holec

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Michal Holec**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Vyhledávání Vzorů v Komprimovaných Datech
Compressed Pattern Matching

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je implementace vybraného algoritmu vyhledávajícího vzory ve zkomprimovaných datech. Program bude schopen zkomprimovat vstupní data zvoleným algoritmem. V takto zkomprimovaných datech bude schopen vyhledat libovolný text.

1. Základní přehled algoritmů pro vyhledávání vzorů v komprimovaných i nekomprimovaných datech.
2. Detailní popis zkoumaného algoritmu.
3. Experimenty a jejich vyhodnocení. Především porovnání velikosti souborů a rychlosti vyhledání vzorků na vstupu.

Seznam doporučené odborné literatury:

[1] Salomon, D., Data Compression: The Complete Reference 4th Edition.
[2] Takeda, M., Encyclopedia of Algorithms: Compressed Pattern Matching, 2008, Springer US.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Michal Vašinek**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. Červen 2019

Michal Bílka

.....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 28. Červen 2019 .....

Rád bych poděkoval vedoucímu práce Ing. Michalovi Vašinkovi za odbornou pomoc, vedení a trpělivost, kterou mi v průběhu zpracování diplomové práce věnoval.

Abstrakt

Tato bakalářská práce se zabývá vyhledáváním vzorků v datech. Hlavním úkolem je popsat vybrané algoritmy a datové struktury, pomocí kterých se takové vyhledávání v praxi provádí a to v datech nekomprimovaných i komprimovaných. Nedílnou součástí tohoto úkolu je i implementace vybrané datové struktury. V současnosti se hojně používají komprimační algoritmy využívající Burrows-Wheelerovu transformaci, na které závisí datová struktura FM-Indexu, kterou budeme implementovat. Implementace je provedena v programovacím jazyce `C#`. Nad výslednou datovou strukturou budou provedeny experimenty, které se zaměří na rychlost vyhledávání a prostorové nároky. Rychlost vyhledávání bude porovnána oproti klasickým algoritmům. Prostorové nároky budou porovnány podle formátu vstupních dat a při různých konfiguracích FM-Indexu. Na závěr jsou prezentovány výsledky a zjištěné poznatky z experimentů implementované datové struktury.

Klíčová slova: Vyhledávání vzorů, Algoritmus Boyer-Moore, Suffixové pole, Transformace Burrows-Wheeler, FM-Index

Abstract

This Bachelor's thesis is about pattern matching. Main objective is to describe selected algorithms and data structures, that are used in practice for pattern matching on non-compressed as well as compressed data. Integral part of this thesis is subsequent implementation of the selected data structure. At present, compression algorithms using Burrows-Wheeler transformation are used extensively and data structure FM-Index depends on it. This data structure will be implemented in programming language `C#` and subjected to experiments. Experiments will mainly cover speed of pattern matching and will be cross examined against more classical algorithms. Space requirements will be tested on data of varying formats as well as with different configurations of FM-Index. At the end the results and findings from the experiments will be presented.

Key Words: Pattern Matching, Boyer-Moore, Suffix array, Burrows-Wheeler Transformation, FM-Index

Obsah

Seznam použitých zkratek a symbolů

KMP	–	Algoritmus Knuth–Morris–Pratt
BM	–	Algoritmus Boyer-Moore
BWT	–	Burrows–Wheeler transformace
`C#`	–	Programovací jazyk C Sharp
MTF	–	Move-to-front kódování

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Vyhledávání vzorů je dlouhodobě řešený problém v počítačové vědě. Jedná se o pokus nalezení sekvence symbolů ve větším bloku dat. První algoritmy řešící tuto problematiky nebyly moc efektivní a bylo nutné vymyslet rychlejší. Postupně vznikaly nové algoritmy, které se snažily využívat informace získané při vyhledávání, anebo analýzou vstupních dat, nebo vyhledávaného vzorku. Tyto algoritmy nakonec daly vzniku algoritmu Boyer-Moore, který se poté stal standardem pro vyhledávání a měřítkem výkonu nových algoritmů.

Tím ale snaha dosáhnout rychlejších algoritmů pro vyhledávání neskončila. Vytvořit nový revoluční algoritmus se ale nedařilo, a tak se vývoj ubíral směrem k pomocným datovým strukturám. Mezi prvními takovými datovými struktura bylo Trie, prefixový strom, ze kterého se později vyvinul sufixový stromu. Obě tyto struktura umožňovali podstatně rychlejší vyhledávání, ale na úkor velkých prostorových nároků. Díky tomu později vznikla datová struktura sufixového pole, což byla kompaktnější verze sufixového stromu. I přestože sufixové pole bylo podstatně menší než sufixový strom, tak stále nabývalo asi čtyři krát větších rozměrů, než vstupní data. Toto byl nepřekonatelný problém v některých oblastech vědy, kde se pracuje s obrovskými daty.

Tento problém vyřešila Burrows–Wheeler transformace. Ta využívá vlastností sufixového pole, zároveň ale pouze přeskládává vstupní data, takže nevyžaduje žádné místo navíc. Dokonce má tu výhodu, že přeskládaná data jsou lepší pro komprimování a tím lze prostorové nároky ještě více snížit. Což bylo také později realizováno jako datová struktura FM-Index.

Hlavním bodem této práce je pospat funkce FM-Indexu a následně tuto datovou strukturu implementovat. K dosažení tohoto bodu je nutné vysvětlit části ze kterých se FM-Index skládá. Proto je nutné popsat postupný vývoj, který vedl k jeho vytvoření.

Bakalářská práce je rozdělena do dvou částí, teoretickou a praktickou. Teoretická část se skládá ze tří částí. V první části jsou popsány algoritmy a datové struktur pro vyhledávání v nekomprimovaných datech. V druhé části je úvod do problematiky vyhledávání v komprimovaných datech. Algoritmy, které využíváme v praktické části, budou popsány v této části. Ve třetí časti je popsána Burrows-Wheelerova transformace a fungování FM-Indexu.

V praktické části se věnujeme vlastní implementaci FM-Indexu. Vlastnostem FM-Indexu a jejich využitím při různých konfiguracích. Problémům vzniklých při implementaci a jejich řešením. Experimenty se skládají z porovnání vlastní implemenetace FM-Indexu proti algoritmu Boyer-Mooru a porovnání různých konfigurací FM-Indexu. Porovnávány budou hlavně rychlosti vyhledávání a pro FM-Index zvláště velikost generovaných dat.

2 Vyhledávání vzorů v textu

V této kapitole bude popsána základní problematika vyhledávání vzorů v textu. Algoritmy a funkce popsané v této kapitole, ale nejsou limitovány pouze na text a je možné je použít i na data jiného formátu.

Vyhledávání se v podstatě skládá ze tří základních operací:

- Operace *Obsahuje* (anglicky *Contains*), jedná se o operaci, která nás informuje, zda se vůbec vzorek v textu nachází.
- Operace *Počet* (anglicky *Count*), tato operace nás informuje kolikrát se daný vzorek v textu nachází.
- Operace *Najít* (anglicky *Find*), operace, jejímž výsledkem je množina indexů, které nám udávají, kde v textu se vzorky nacházejí.

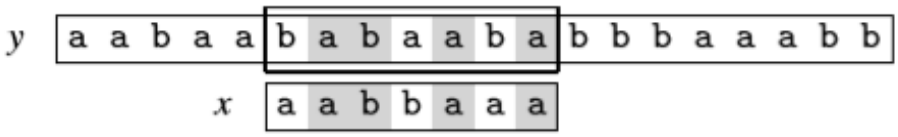
Nechť je T textem, ve kterém vyhledáváme. Pro zjednodušení lze uvažovat o operaci *Obsahuje* jako o vyhodnocení $Contains = Count(T) > 0$. O operaci *Počet* jako o velikosti výsledné množiny operace *Najít*, $Count = Length(Find(T))$.

Každá z těchto operací může být jinak implementována a díky tomu být rychlejší. Například pro operaci *Obsahuje* stačí zjistit, jestli se vzorek v textu T vyskytuje alespoň jednou a není nutné zjišťovat přesný počet a hledat další výskyt vzorku. Z tohoto důvodu mohou mít některé algoritmy i operaci pro nalezení prvního výskytu. Operace *První* (anglicky *First*), kde je vracen index prvního výskytu vzorku v textu T. Tato operace bude uvedena v některých ukázkových algoritmech. Lze o ní uvažovat jako o zkrácené operaci *Najít*, která se pouze liší tím, že dříve skončí a vrátí výsledek. Rozšíření na plnohodnotnou operaci *Najít* je potom pouze o zapsaní výsledku do bufferu a pokračování v algoritmu.

Je nutné zmínit, že některé algoritmy mají tyto operace svázány, což znamená, že jejich rychlost bude stejná. U všech níže popsaných algoritmů v této kapitole lze předpokládát, že operace *Počet*, $Count = Length(Find(T))$, bude stejně náročná jako operace *Najít*. To plyne z podstaty jak tyto algoritmy fungují a nelze je nijak výrazněji optimalizovat. Toto však neplatí pro indexovací datové struktury, kde operace *Počet* je velmi rychlá, ale operace *Najít* může být podstatně náročnější, pokud chceme snížit prostorové nároky.

2.1 Koncept posuvného okna

Koncept posuvného okna je společný pro mnoho vyhledávacích algoritmů. Jestliže vzorek není prázdná sekvence znaků, tak je vhodné při vyhledávání v textu T do něj nahlížet pomocí takzvaného posuvného okna. Okno vymezuje část textu, která se nazývá obsah okna. Obsah okna má ve většině případů délku stejnou jako vyhledávaný vzorek. Při vyhledávání se okno postupně posouvá podél textu od začátku do konce.



Obrázek 1: Ilustrace posuvného okna, pokus nalézt vzorek $x = aabbaaa$ v textu $y = aabaababaabablbbaaabb$ při posunu $i = 5$

Okno se nachází v dané pozici i v textu a algoritmus testuje, zda se vzorek P vyskytuje ve stejné pozici. Provádí to tak, že porovná některé znaky obsahu okna se zarovnanými znaky v textu T. Tato operace se nazývá pokus o nález na pozici i. Pokud je porovnání úspěšné, tak se podařilo najít výskyt vzorku P v textu T na pozici i. Během této fáze testu algoritmus získává informace o textu, které může následně využít a to buď nastavením posunu okna i, anebo si zapamatuje získané informace, aby se vyhnul stejnému porovnávání při dalších pokusech.

Když je posun proveden z polohy i do polohy $i+j(j \geq 1)$, bývá tento jev označován jako posun délky j. Posun j musí být platný, což znamená, že pokud je posun $j \geq 2$, tak se vyhledávaný vzorek nesmí nacházet na pozici $i + 1$, což je stejné jako $i + j - 1$ v textu. Pokud by daný jev nastal, tak by došlo k přeskočení nálezu a daný algoritmus by nebyl korektní. [1]

Pro zjednodušení výkladu níže uvedených algoritmů budou v této kapitole jednotně označovány následující proměnné: vyhledávaný vzor jako $P[1, p]$, jehož délka je p a vstupní texte, ve kterém bude vyhledáváno, jako $T[1, u]$ o délce u. Bude předpokládáno, že platí $p < u$ jinak by se vzorek P v textu T logicky nemohl nacházet.

Koncept posuvného okna ilustruje obrázek 1. [1], strana 28.

2.2 Algoritmy pro vyhledávání v textu

V této podkapitole jsou popsány algoritmy zabývající se vyhledávání vzorků v textu. Počínaje od nejjednodušších, po sofistikovanější. Díky svým rozdílným implementacím se i jejich efektivita co se týče rychlosti vyhledávání, nutnosti nějakým způsobem si přichystat data před samotným vyhledáváním, anebo prostorovými nároky liší.

2.2.1 Algoritmus Naivního vyhledávání v textu

Naivní algoritmus pro vyhledávání v textu, je nejjednodušší algoritmus pro vyhledávání vzorů. Jako jediný si před samotným vyhledáváním nezpracovává vzorek, ani text ve kterém vyhledává.

Nejjednodušší implementace posouvání okna a porovnávání je uvedena v algoritmu pro naivní vyhledávání, viz Algoritmus 1. Po každé operaci porovnávání je posunuto okno o jednu pozici doprava, což zřejmě generuje správný algoritmus, pokud porovnání nejsou špatně implementována. V algoritmu se používá proměnná i, která indikuje aktuální pozici okna a proměnná

j, která udává pozici porovnávaného znaku v posuvném okně. Znak textu $T[i+j]$ je tak tedy vždy porovná se symbolem $P[j]$.[1]

Díky své podstatě algoritmus nepotřebuje předem zpracovávat vzorek, ani text a nepotřebuje žádné místo navíc. Jeho nevýhodou je efektivita vyhledávání. Jak lze vypozorovat, tak algoritmus nijak nevyužívá informace z předchozích porovnávání a proto může redudantně porovnávat znaky, které již předtím porovnával.

Algorithm 1 Algoritmus pro naivní vyhledávání

```
1:  function Naive_Search(P, T, p, u)
2:      i = 0;
3:      while (i ≤ u − p) do
4:          j = 0;
5:          while (j < p and T[i + j] = P[j]) do
6:              j = j + 1;
7:          end while
8:          if j ≥ p then
9:              return "match found at position (i)";
10:         end if
11:         i = i + 1;
12:     end while
13:     return "no matches";
14: end function
```

2.2.2 Algoritmus Knuth–Morris–Pratt

Tento algoritmus vytvořili a spolu publikovali Donald Knuth, Vaughan Pratt a James H. Morris v roce 1977. [2]

Je to první algoritmus, který byl schopný vyhledávat vzory v lineárním čase a zároveň je velmi podobný algoritmu naivního vyhledávání. Po bližší analýze algoritmu naivního vyhledávání se snaží napravit plýtvání informacemi, které jsou zjištěny při pokusech porovnání. Ze které vyplívá, že je možné zlepšit délku posunů a zároveň zaznamenat některé části textu, které se shodují s částí vyhledávaného vzorku. Napravením těchto nedostatků pak nedochází k opětovnému porovnání stejných znaků a zvyšuje se tím rychlost vyhledávání.

Předpokládejme, že při pokusu porovnání došlo k neshodě na pozici $i > 0$ a zároveň $j > 0$. Když dochází k posunu, tak lze předpokládat, že prefix vzorku se shoduje s částí textu, kterou jsme už porovnávali. Posun tedy uděláme o j - $A[j]$, kde A je předpřipravená vyhledávací tabulka obsahující hodnotu upravující posun, tak aby nedošlo k přeskočení možného nálezu. Podle této tabulky také nastavujeme j. Není zapotřebí začínat porovnávat znaky v okně od 0, protože prvních j znaků je stejných. [3]

2.2.2.1 Příprava pomocné vyhledávací tabulky Pro vyhledávací tabulku $A[1,p]$ potřebujeme stejnou velikost jakou má vzorek P, proto teto algoritmus potřebuje navíc prostor o

Tabulka 1: Pomocná vyhledávací tabulka pro KMP sestavená pro vzor "ban a banana"

Index (j)	0	1	2	3	4	5	6	7	8	9	10	11
Vzorek	b	a	n		a		b	a	n	a	n	a
Posun	-1	0	0	0	0	0	-1	0	0	3	0	0

velikosti O(u + p). První hodnota $A[0]$ je nastavena na -1. Posun je počítán jako $j - A[j]$ a je nutné, aby výsledek byl vždy kladný. Pro zbylé hodnoty je nastavena na 0. Poté v cyklu projdeme celý vzorek P a do tabulky se zapíše délka sekvence znaků, které jsou stejné jako začátek vzorku.

Algoritmus pro sestavení vyhledávací tabulky lze vidět na Algoritmu 2.

Algorithm 2 Algoritmus pro sestavení pomocné vyhledávací tabulky pro KMP

```
1:  procedure KMP_PrepareTable(P, p, A)
2:      i = 0;
3:      j = −1;
4:      A[0] = −1;
5:      while (i < p) do
6:          while j > −1 and P[i] ≠ P[j] do
7:              j = A[j];
8:          end while
9:          i = i + 1;
10:         j = j + 1;
11:         if P[i] = P[j] then
12:             A[i] = A[j];
13:         else
14:             A[i] = j;
15:         end if
16:     end while
17: end procedure
```

Jak takto sestavená tabulka pro vzorek "ban a banana"vypadá, lze vidět v tabulce 1.

2.2.2.2 Vyhledávání pomocí KMP Vyhledávání probíhá podobně jako algoritmus pro naivní vyhledávání. Před samotným vyhledáváním je sestavena pomocná tabulka A, pomocí které lze určit správný posun, tak aby nebyl minut nějaký výskyt vzoru P a zároveň aby se nevykonávaly některá porovnání zbytečně několikrát.

Posun je vypočítáván jako $i = i + j - A[j]$, kde i je základní posun okna a j je posun v obsahu okna. Ukázková implementace je znázorněna na Algoritmu 3

2.2.3 Algoritmus Boyer–Moore

Algoritmus vytvořili Robert S. Boyer a J Strother Moore v roce 1977 a stal se standardním měřítkem pro porovnávání algoritmů pro vyhledávání v textu. [7]

Algorithm 3 Vyhledávací algoritmus Knuth-Morris-Pratt

```
1: function KMP_Search(P, T, p, u)
2:     i = 0;
3:     j = 0;
4:     A = [0...p];
5:     KMP_PrepareTable(P, p, A);
6:     P = [...];
7:     iP = 0;
8:     while i < u do
9:         if P[j] = T[i] then
10:            i = i + 1;
11:            j = j + 1;
12:            if j = p then
13:                P[iP] = i − j;
14:                iP = iP + 1;
15:                j = A[j];
16:            end if
17:        else
18:            j = A[j];
19:            if j < 0 then
20:                i = i + 1;
21:                j = j + 1;
22:            end if
23:        end if
24:    end while
25:    return P;
26: end function
```

Tabulka 2: Tabulka pravidla špatného znaku pro vzor "banana"

Znak	a	b	n	ostatní znaky
Posun	0	5	1	6

Tento algoritmus si před samotným vyhledáváním zpracuje vyhledávaný vzorek P a pomocí dvou pravidel si sestaví pomocné vyhledávací tabulky. Funguje v podobném duchu jako algoritmus Knuth-Morris-Pratt a snaží se přeskakovat znaky, u kterých ví, že je nemusí porovnávat. Jednou z hlavních vlastností tohoto algoritmu, kterou se liší od ostatních, je že porovnávání vzorku provádí odzadu. Text stále procházen zleva doprava, ale samotné porovnávání v obsahu posuvného okna se provádí zprava doleva.

2.2.3.1 Zpracování vzoru Před samotným vyhledáváním si algoritmus pomocí dvou pravidel sestaví vyhledávací tabulky. Tyto tabulky následně využívá při vyhledávání a určuje pomocí nich maximální posun. Tyto pravidla jsou jednak pravidlo špatného znaku, tak pravidlo dobré přípony. Tím, že tyto pravidla a jejich tabulky jsou dvě, tak jsou i dvě možné hodnoty posunu, přičemž je vždy vybírána ta vyšší hodnota.

2.2.3.2 Pravidlo špatného znaku Toto pravidlo využívá situace, kdy se při vyhledávání porovnávané znaky neshodují. V tom případě se zjistí, jestli se porovnávaný znak textu T na pozici $i + j$ nachází ve vzorku a pokud ano, tak je proveden posun tak, aby nejpravější výskyt tohoto znaku ve vzorku P byl zarovnán s obsahem okna a výskytem stejného nejpravějšího znaku. Pokud se daný znak v P nenachází, tak je posun proveden o celou délku P tak, aby okno začínalo až za následujícím znakem v T.

Toto pravidlo je řešeno tak, že je vytvořena vyhledávací tabulka a analyzován vzorek P. Pro každý znak vzorku P je zapsán do tabulky nejpravějšího výskytu tohoto znaku od konce vzorku P. Pro znaky, které se v abecedě vzorku P nevyskytují, tabulka vrací délku vzorku P.

Algoritmus pro vytvoření takovéto tabulku, lze vidět na Algoritmu 4. Výsledná vzorová tabulka lze vidět v ukázkové Tabulce 2.

Algorithm 4 Algoritmus pro vytvoření vyhledávací tabulky pro pravidlo špatného znaku

```
1: function BM_PrepareBadCharTable(P, p)
2:     badCharTable = [...];
3:     badCharTable[other] = p;
4:     i = 0;
5:     while i < p do
6:         badCharTable[P[i]] = p − i − 1;
7:         i = i + 1;
8:     end while
9:     return badCharTable;
10: end function
```

Tabulka 3: Tabulka pravidla dobré přípony pro vzor "banana"

Index	0	1	2	3	4	5
Vzor	b	a	n	a	n	a
Posun	11	10	5	8	5	1

2.2.3.3 Pravidlo dobré přípony Toto pravidlo využívá situace, kdy se část obsahu okna shoduje a zjišťuje, jestli se ve vzorku P nenachází stejná sekvence shodujících znaků. Pokud ano, tak je ve vyhledávací tabulce uveden takový posun, aby se okno zarovnalo na pozici, kde se shodující sekvence znaků překrývají.

Příklad takto sestavené tabulky je na ukázkové Tabulce 3

2.2.3.4 Vyhledávání pomocí BM algoritmu Po zpracování vzorku P algoritmus prochází celý text T a posunuje okno zleva doprava, posun okna je dán proměnou i. Algoritmus porovnává znaky s obsahem posuvného okna a to zprava doleva. Pro určení pozice v obsahu okna je využívána pomocná proměnná j, jejíž hodnota začíná na hodnotě délky vzorku P a je postupně snižována. Pokud j dosáhne 0, tak to znamená, že se veškeré znaky v okně shodovali a hledaný vzorek byl nalezen. Pokud dojde při porovnávání k neshodě, tak je vypočítán posun pro i a to tak, že si vezmeme větší posun z předchystaných tabulek. Posun z tabulky vytvořené pomocí pravidla špatného znaku je dán podle znaku, který se nachází v textu T na pozici $i + j$. Z tabulky pro dobré přípony si vezmeme posun podle pozici j. K i přičteme větší z těchto dvou hodnot a j nastavíme na délku vzorku$P$.

Na ukázkový algoritmus pro nalezení prvního výskytu se lze podívat v Algoritmu 5

Algorithm 5 Vyhledávací algoritmus BM

```
1:  function BM_Search(P, T, p, u)
2:      badCharTable = BM_PrepareBadCharTable(P, p);
3:      goodSuffixTable = BM_PrepareGoodSuffixTable(P, p);
4:      i = 0;
5:      while i ≤ u − p do
6:          j = p − 1;
7:          while j ≥ 0 and P[j] = T[i + j] do
8:              if j = 0 then
9:                  return "match found at (i)";
10:             end if
11:             j = j − 1;
12:         end while
13:         i = i + max(badCharTable[T[i + j]], goodSuffixTable[j]);
14:     end while
15:     return "no match found";
16: end function
```

2.3 Indexovací datové struktury

V této části jsou popsány datové struktury používané k rychlejšímu vyhledávání vzorků v textu. Začínaje sufixovým stromem a kompaktnější verzí této datové struktury, sufixovým polem. Obě tyto struktury vznikly, aby bylo dosaženo rychlejšího vyhledávání než nám umožnují klasické algoritmy řešící problematiku vyhledávání v textu. I přesto že jsou obě datové struktury podstatně rychlejší, tak jejich využití je omezené z důvodů velkých prostorových nároků.

2.3.1 Sufixový strom

Sufixový strom, také nazýván PAT strom, je datová struktura sloužící k prezentaci množiny všech sufixů daného textu. Tato datová struktura je velmi podobná datové struktuře Trie, ale oproti Trie využívá podstatně méně prostoru. Zatímco velikost Trie může být až kvadratická v závislosti na velikosti vstupního řetězce, sufixový strom poskytuje lineární reprezentaci prostoru. Sufixový strom je efektivní jak v čase vyhledávání, tak v prostoru a je využíván pro různé aplikace.

Aby bylo možné jednoduše popsat vlastnosti suffixového stromu, tak je nutné definovat několik proměnných. Představme si řetězec T o délce u, který je rozšířen o symbol označující konec řetězce. Tento symbol se nachází na konci řetezce a nelze jej vidět, většinou se označuje jako \$, celková délka je tedy $u + 1$. Sufixový strom takového řetězce $T\$$ má poté následující vlastnosti:

1. Přesně $u + 1$ listových uzlů.
2. Nanejvýš u vnitřních uzlů (kořenový uzel je brán jako vnitřní uzel).
3. Každý unikátní podřetězec řetězce T je zakódován v sufixovém stromě přesně jednou. Při cestě z kořenového adresáře do libovolného uzlu n tak vznikne podřetězec $L(n)$, který je unikátní a nachází se ve stromě přesně jednou.
4. Žádný z poduzlů jednoho uzlu nesmí začínat stejným znakem.
5. Každý vnitřní uzel má minimálně 2 pod uzly.

Z vlastností 1, 2, 4 a 5 vyplívá, že sufixový strom bude mít celkově nanejvýš $2u + 1$ uzlů a nanejvýš $2u$ listových uzlů. Sufixový strom je tedy velmi podobný tradiční datové struktuře Trie. Největší rozdíl je, že sufixový strom využívá kompresi okrajů a kompresi cest z kořenů až po okraje. Vzniká tedy taková kompaktnější sufixová Trie. Obě tyto komprese jsou kritické, aby při konstrukci sufixového stromu bylo dosaženo vyhledávání v lineárním čase a lineárních prostorových nároků.

Konstrukce sufixového stromu není složitá. Jednoduchý algoritmus lze nalézt jako Algoritmus 6, tento algoritmus splní svůj účel, ale není velice efektivní. [8], strana 51- 55

Algorithm 6 Jednoduchý algoritmus pro konstrukci Sufixového stromu

```
1: function SIMPLE-SUFFIX-TREE-ALGORITH(T)
2:     Vytvoření kořenového uzlu root, s prázdným řetězcem
3:     for i ← 1 to n do
4:         Projdi strom z kořenového uzlu root
5:         Po jednoom porovnej symboly v krajních uzlech se symboly momentálního sufixu, T_i
6:         if dojde k neshodě then
7:             Rozděl krajní uzel v pozici kde došlo k neshodě a pokud je to nutné, tak vytvoř
   nový uzel
8:             Vlož sufix T_i do sufixového stromu v pozici kde došlo k neshodě
9:         end if
10:    end for
11: end function
```

2.3.2 Sufixové pole

Stejně důležitá datová struktura, která úzce souvisí se sufixovými stromy, je sufixové pole. Sufixové pole poskytuje abecedně seřazený seznam všech sufixů vstupního řetězce. Pokud se prvek v sufixovém poli nachází na pozici i a má hodnotu j, tak to znamená, že suffix začínající na pozici j v řetězci T je i-tý nejmenší suffix nacházející se v celém řetězci T.

Pokud jsou k dispozici informace o celkovém počtu jednotlivých symbolů textu T. Tak kombinace těchto informací spolu se sufixovým polem poskytuje výkonnou datovou strukturu pro vyhledávání. Zjištění počet výskytů vzorku P s délkou p v řetězci T o délce u je poté možné provést v čase $O(p + \log u)$.

Sufixové pole lze využívat ve většině případů, kde se používá sufixový strom, ale ne ve všech. V těchto případech je možné sestavit sufixový strom ze sufixového pole a to v lineárním čase. Hlavní motivací, proč jsou využívány sufixová pole oproti sufixovým stromům je jejich úspora prostoru. Teoreticky je prostorová složitost pro obě datové struktury lineární. Nicméně sufixová pole potřebují asi 3 krát až 5 krát méně prostoru než sufixové stromy.

Konstrukční čas pro oba algoritmy je průměrně $O(u)$. Algoritmy pro sufixová pole, které běží v nejhorším případě $O(u \log u)$, jsou poměrně jednoduché na vytvoření, ale algoritmy, které běží v nejhorším případě $O(u)$ jsou naopak mnohem pracnější na vytvoření. Tradičně pak platí, že vytvoření sufixového pole trvá déle než vytvoření sufixového stromu.

Jednoduchý způsob, jak sestavit sufixové pole je představit si podřetězce jako vektory. Každý vektor odpovídá jednomu sufixu z textu T. Tyto vektory lze potom setřídit pomocí klasických třídících algoritmů, například pomocí quicksort. Vždy jsou vybrány dva vektory a porovnány podřetězce, na které se odkazují. Teoreticky lze ale sufixové pole vytvořit rychleji pomocí sufixového stromu. Pokud je sufixový strom k dispozici, tak lze projitím celého stromu vytvořit sufixové pole v lineárním čase. Tento úkon se například provádí pomocí Farachovi rekurzivní konstrukce, kde jsou všechny krajní uzly seřazeny pomocí poduzlů. Z toho lze usoudit, že pokud jsou všechny koncové uzly seřazeny abecedně zleva doprava, tak stačí provést hloubkový průchod

sufixového stromu, který sestaví sufixové pole. Vzhledem k požadavkům na snížení prostoru sufixových stromů je zájem o přímou konstrukci sufixových polí, aniž by bylo nutné sestavit sufixový strom.

3 Vyhledávání vzorků v komprimovaných datech

Vyhledávání vzorů v komprimovaných datech je snaha nalézt všechny výskyty vzoru v komprimovaných datech a to bez toho, aby byly tyto data vůbec dekomprimovány, anebo byly dekomprimovány celé. Tento problém představili poprvé Amir A., Benson G. a Farach M. ve své práci jako úkol vyhledat vzorky v komprimovaných datech bez toho, aby je dekomprimovali. Hlavními výhodami vyhledávání v komprimovaných datech je úspora prostoru a ve většině případech i rychlejší vyhledávání. [4]

3.1 Move-to-front kódování

Move-to-front kódování, ve zkratce MTF, je metoda, která mění symboly vstupních dat za počet předešlých symbolů v abecedě těchto dat. Pro tuto metodu existuje několik variant a nejpopulárnější z nich přidává jeden krok navíc: poté co je symbol zakódován, je v abecedě přesunut na první pozici. Je očekáváno, že se symboly ve vstupních datech budou opakovat. Pokud je toto očekávání splněno a data obsahují běhy opakujících se znaků, tak nám tato technika umožní data lépe komprimovat. [11]

3.1.1 MTF Kódování

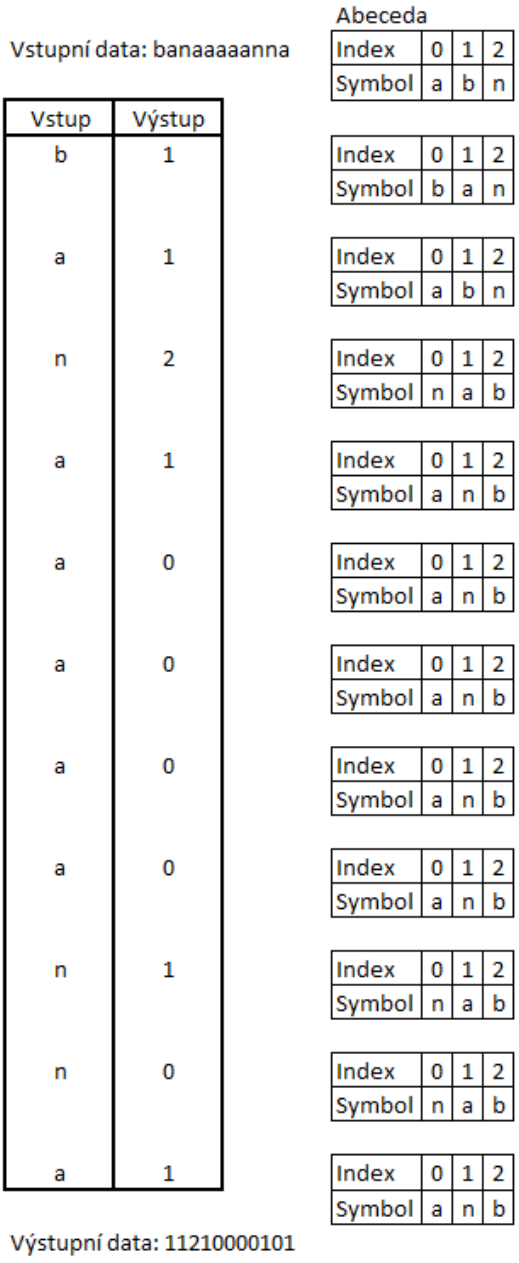
Na začátku kódování je sestavena a seřazena abeceda vstupních dat.

Kódování potom probíhá tak, že každý symbol, který se nachází ve vstupních datech je zakódován jako počet předešlých symbolů v abecedě. Tato abeceda není pevně dána a při kódování se pořadí symbolů postupně mění. Pokaždé když je nějaký symbol zakódován, tak je přesunut v abecedě na její začátek. Všechny opakující se symboly jsou následně zakódovány jako 0. Abeceda zůstává stejná, protože se symbol nezměnil.

V případě, že vstupní data budou převážně obsahovat opakující se znaky, tak výstupní data budou tvořit převážně nuly. Pokud je toto očekávání naplněno, tak budou data lépe komprimovatelná.

Jako příklad si představme, že se na vstupu nachází řetězec "banana". Abeceda tohoto řetězce je $[a, b, n]$. Při zakódování symbolu "b", by byla na výstup zapsána 1 (počet předešlých symbolů v abecedě) a abeceda by se změnila na $[b, a, n]$. Kdyby se symbol "b"bezprostředně znovu opakovalo, tak by byla zapsána 0 a abeceda se nezměnila. Příklad kódování lze vidět na Obrázku 2.

Jak lze vidět, tak se stav abecedy během kódování mění, nejedná se o blokové kódování. Proto nelze kódovat, anebo dekódovat z libovolného místa v řetězci. Aby bylo něco takového proveditelné, tak je potřeba vědět v jakém stavu se abeceda na dané pozici nachází. Při implementaci FM-Indexu byl do každého bloku také zapsán stav abecedy, v jakém se nachází na začátku daného bloku.



Obrázek 2: Ukázka Move-to-front kódování řetězce "banaaaaanna"a toho jak se mění abeceda

3.1.2 MTF Dekódování

Aby bylo možné data dekódovat zpět do původní podoby, tak je zapotřebí si abecedu uložit. Dekódování je poté prováděno tak, že si zjistíme, jaký symbol se nachází v abecedě na daném indexu a ten zapíše na výstup a přesuneme tento symbol v abecedě na začátek, úplně stejně jako při kódování. Dekódování je možné provádět i od konce, ale je k tomu zapotřebí stavu abecedy, ve kterém se nacházela po ukončení kódování. Na výstup je poté zapsán znak, který se nachází na začátku abecedy a přesouváme ho na index, který je v zakódovaných datech.

3.2 Huffmanovo kódování

Huffmanovo kódování je jedna z mnoha statistických kompresních metod. Klasické kódovací metody, jako je i například MTF kódování, nastavují symbolům kódy, které mají stejnou velikost. Statistické metody používají kódy proměnlivých délek a to v závislosti na pravděpodobnostních symbolů. Symbolům, které se v datech vyskytují častěji jsou přiděleny kratší kódy a naopak symbolům s malým počtem výskytů kódy delší.

Huffmanova metoda je poněkud podobná Shannon-Fanově metodě. Obecně produkuje lepší kódy a jako Shannon-Fanova metoda produkuje nejlepší kódy, pokud pravděpodobnosti jednotlivých symbolů jsou negativní mocniny dvojky (0.5, 0.25, 0.125, ...). Hlavní rozdíl mezi oběma metodami je, že Shannon-Fanova metoda vytváří kódy z vrchu dolů, zatímco Huffmanovo kódování vytváří kódy jako strom, který je sestaven ze spodu nahoru. Od momentu svého vyvinutí v roce 1952 D. Huffmanem byla tato metoda intenzivně zkoumána v oblasti data komprese. [11].

3.2.1 Konstrukce Huffmanova stromu a kódování

Algoritmus začíná sestavením seznamu, který se skládá z abecedy vstupních dat a jednotlivé symboly seřadí pole pravděpodobností jejich výskytů, od nejpravděpodobnějších po méně pravděpodobné. Podle těchto pravděpodobností je následně vytvořen strom, se symbolem na každém listovém uzlu a to ze zdola nahoru. Vytváření se provádí v krocích, kdy v každém kroku jsou vybrány 2 symboly s nejmenší pravděpodobností a jsou přidány k horní části dílčího stromu, odstraněny ze seznamu a nahrazeny pomocným symbolem, který představuje oba původní symboly. Jakmile seznam obsahuje pouze jediný pomocný symbol, který reprezentuje celou abecedu, tak je strom kompletní. Celý strom se poté projde a určí se kódy jednotlivých symbolů.

Proces konstrukce Huffmanova stromu lze nejlépe ukázat na příkladu. Představme si, že máme data, která se skládají z abecedy tvořené pěti symboly $[a_1, a_2, a_3, a_4, a_5]$. Pravděpodobnosti těchto symbolů jsou uvedeny v Tabulce 4.

Konstrukce je poté prováděna v následujících krocích:

1. Jsou vybrány symboly a_4 a a_5, jakožto symboly s nejmenší pravděpodobností. Oba symboly jsou zkombinovány do jednoho pomocného symbolu a_{45}, jehož pravděpodobnost je 0,2

Tabulka 4: Pravděpodobnostní tabulka symbolů abecedy $[a_1, a_2, a_3, a_4, a_5]$

Symbol	Pravděpodobnost
a_1	0,4 (40%)
a_2	0,2 (20%)
a_3	0,2 (20%)
a_4	0,1 (10%)
a_5	0,1 (10%)

a tímto symbolem jsou symboly a_4 a a_5 v abecedě nahrazeny. Abeceda je změněna na $[a_1, a_2, a_3, a_{45}]$.

2. V abecedě zůstávají 4 symboly, a_1 s pravděpodobností 0,4 a symboly a_2, a_3, a_{45}, každý s pravděpodobností 0,2. Náhodně jsou vybrány symboly a_3 a a_{45}, které jsou zkombinovány do jednoho symbolu a_{345}, jehož pravděpodobnost je 0,4 a tímto symbolem jsou v abecedě nahrazeny zkombinované symboly.

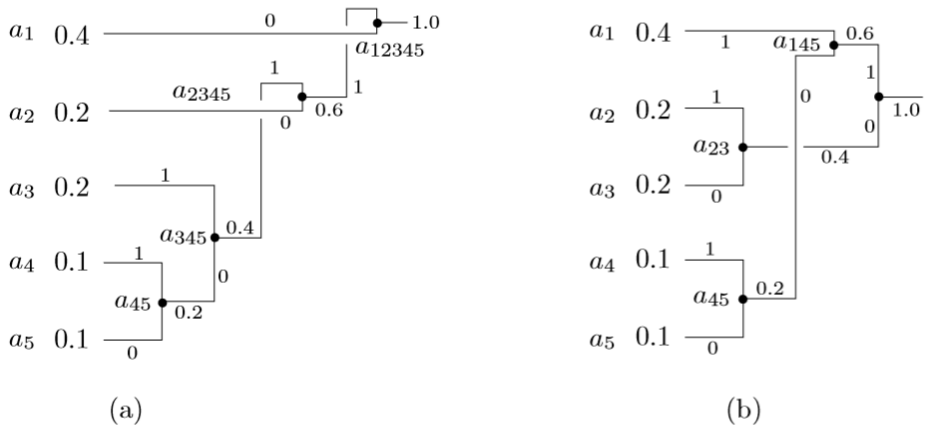
3. V abecedě zůstávají 3 symboly, a_1, a_2, a_{345}, jejichž pravděpodobnosti jsou 0,4, 0,2 a 0,4. Náhodně jsou vybrány symboly a_a a a_{345}, které jsou zkombinovány do jednoho symbolu a_{2345}, jehož pravděpodobnost je 0,6 a tímto symbolem jsou zkombinované symboly v abecedě nahrazeny.

4. V abecedě zůstávají pouze 2 symboly, a_1 a a_{2345}. Oba jsou zkombinovány a nahrazeny symbolem a_{12345}, který má pravděpodobnost 1.

Strom je nyní kompletní a na Obrázku 3a lze vidět, jak vypadá položený na stranu, tak aby kořen byl vpravo a listové uzly nalevo. K přiřazení kódů projdeme celý strom. Budeme postupovat od kořene a náhodně vybereme horní větev stromu, které přiřadíme bitovou hodnotu 1. Spodní větvi přiřadíme bitovou hodnotu 0, z tohoto vzniknou kódy: $0, 10, 111, 1101, 1100$. Toto přiřazení je čistě náhodné a v konečném důsledku nemá na nic vliv.

Průměrnou velikost tohoto kódu lze spočítat jako $0,4 \times 1 + 0,2 \times 2 + 0,2 \times 3 + 0,1 \times 4 + 0,1 \times 4 = 2,2$ bitů na symbol. Jak si lze povšimnou, tak při konstrukci byla některá rozhodnutí provedena náhodně, z toho lze vyvodit, že Huffmanovo kódování není unikátní. Protože při sestavování symbolů nastala situace, kdy byly v seznamu více než dva symboly se stejnou nejnišší pravděpodobností, je možné vyzkoušet jiný způsob zkombinování symbolů. Na Obrázku 3b lze vidět jinak sestavený strom nad stejnou abecedou. Průměrná velikost tohoto kódu je $0,4 \times 2 + 0,2 \times 2 + 0,2 \times 2 + 0,1 \times 3 + 0,1 \times 3 = 2,2$ bitů na symbol, tedy stejné jako v minulém případě.

Z tohoto si lze vyvodit, že náhodná rozhodnutí při sestavování stromu nemají vliv na konečnou průměrnou velikost. Vyvstává zde otázka, které kódování je lepší. Odpověď je jednoduchá. Nejlepší kódy jsou ty, které mají nejmenší proměnlivost. Proměnlivost kódu lze určit jako rozdíl velikostí kódu individuálních symbolů oproti průměrné velikosti. Proměnlivost kódu z Obrázku 3a je $0,4 \times (1-2,2)^2 + \times (2-2,2)^2 + 0,2 \times (3-2,2)^2 + 0,1 \times (4-2,2)^2 + 0,1 \times (4-2,2)^2 = 1,36$.



Obrázek 3: Ukázka stromu pro Huffmanovo kódování, oba stromy jsou vytvořeny nad stejnou abecedou se stejnými pravděpodobnostmi

Zatím co proměnlivost kódu z Obrázku 3b je $0,4 \times (2-2,2)^2 + \times(2-2,2)^2 + 0,2 \times (2-2,2)^2 + 0,1 \times (3-2,2)^2 + 0,1 \times (3-2,2)^2 = 0,16$.

Kód stromu z Obrázku 3b je tedy preferovanější. Při pohledu na oba stromy je možné si všimnou, že místo kombinace a_{234} je provedena kombinace a_{145}. Z toho vyplívá následující pravidlo. Pokud jsou k dispozici více než dva symboly s nejnižší pravděpodobností, tak zvolíme jeden symbol s nejnižší a jeden symbol s nejvyšší pravděpodobností. Tato kombinace zkombinuje symboly s nejnižší pravděpodobností spolu se symboly s nejvyšší pravděpodobností, díky čemuž poté dochází k redukci proměnlivosti kódu.

Kdyby kodér pouze zapisoval zkomprimovaná data do souboru, tak je jedno, který kód se použije. Kód s menší proměnlivostí je vhodný pouze pro případy, že kodér přenáší data přes nějakou komunikační linku v momentě, kdy jsou generována. Kodér s velkou proměnlivostí by generoval symboly nepravidelněji, a protože bity musí být přenášeny konstantní rychlostí, tak by bylo zapotřebí většího zásobníku. [11], strany 74 - 76.

3.2.2 Dekódování

Před zahájením dekódování je potřeba zjistit kódy, které mají jednotlivé symboly přiřazeny. Toho lze dosáhnout stejně jako při konstrukci stromu, stačí znát abecedu a pravděpodobnosti jednotlivých symbolů. V praxi to je většinou řešeno tak, že před zakódovaná data zapíšeme všechny symboly abecedy a počet jejich výskytů. Této oblasti se většinou říká hlavička a je uvedena na začátku před zakódovanými daty tak, aby každý Huffmanový dekodér byl schopný si strom a kódy jednotlivých symbolů znovu sestavit. Tento zápis většinou není zakódován, protože jeho velikost je malá a na výstup přidá jenom několik stovek bytů.

Je také možné zapsat kódy přímo na výstup. To může být ale problém, protože kódy mají proměnlivou velikost. Jiný způsob je i zapsání samotného stromu, ale takové řešení většinou

zabere více místa než jen pouhé zapsání symbolů a jejich frekvencí výskytu.

Aby bylo možné dekódovat data, tak je potřeba, aby na začátku dat byly data pro sestavení Huffmanova stromu pro danou abecedu, pouze poté lze dekódovat zbytek dat. Algoritmus pro dekódování je jednoduchý. Začínáme v kořeni stromu a postupně čteme bity ze vstupu, podle toho volíme větev, po které ve stromě půjdeme a to až do momentu, než se dostaneme na listový uzel. Který bit je pro kterou větev si můžeme určit, ale potom by takový dekodér fungoval pouze pro specifický kodér, proto se pro spodní větev používá bit 0 a pro horní větev bit 1. V momentě kdy se dostaneme na listový uzel, tak dekodér zapíše na výstup tento symbol a čte další data s tím, že pokaždém zápisu začne procházet strom znovu od kořene. Toto se opakuje, dokud nejsou dekódována všechna data.

4 FM-Index

4.1 Transformace Burrows-Wheeler

Burrows-Wheelerovu transformaci vymysleli Michael Burrows a David Wheeler v roce 1994 a je založena na původní nepublikované transformaci, kterou objevil D. Wheeler už v roce 1978. Udělat praktické řešení se mu podařilo až s pomocí M. Burrowse v roce 1994. Svůj výzkum publikovali ve výzkumném centru digitálních systémů (Palo Alto), výzkumná zpráva (Burrows a Wheeler, 1994). [8, 9]

Burrows-Wheelerova transformace, dále jen BWT, vytváří permutaci vstupního textu a neposkytuje žádnou kompresi. Teoreticky je prováděna tak, že jsou vytvořeny a abecedně seřazeny všechny sufixy vstupního textu. Předešlý znak těchto sufixů je zapsán do výstupní permutace. V praxi by vytvoření všech sufixů zabíralo velké množství prostoru a pro delší texty by nebylo realizovatelné. Proto je praxi konstrukce BWT většinou řešena pomocí sufixového pole.

Výhoda celé transformace je poté postavena na dvou vlastnostech. Všechny sufixy vstupního textu jsou abecedně seřazeny a stejné sufixy májí většinou stejný prefix. Díky této vlastnosti jsou ve výstupní permutaci běhy stejných znaků a text je jednodušší zkomprimovat.

BWT má taktéž tu výhodu že výsledný text je možné převést do původního tvaru a nejsou k tomu potřeba žádná pomocná data. Zapotřebí je pouze ukazatel na konec původního řetězce. Některé algoritmy na konec vstupního textu přidávají speciální znak, díky kterému je tento ukazatel pak obsažen ve výsledné permutaci jako index odkazující na tento speciálního znaku.

Komprimační algoritmy postavené na bázi BWT patří k nejlepším dostupným, protože dosahují velmi dobrého kompresního poměru za použití relativně malého času a prostoru.

4.2 FM-Index

FM-Index patří mezi indexovací datové struktury, které pracují se zkomprimovanými daty. Je to oportunistická datová struktura, protože její velikost je závislá na tom, zda jsou vstupní data komprimovatelná. Potřebuje méně prostoru a této redukce prostoru dosahuje bez výrazného zpomalení dotazů na vyhledávání. V případě, že jsou data nekomprimovatelná, tak v současnosti dosahuje nejlepších velikostí. Na komprimovatelných datech tato struktura funguje jako vylepšené sufixové pole a to buď ve velikosti prostoru, rychlosti vyhledávání, anebo v obojím.

Dlouho se věřilo, že sufixové pole jsou nekomprimovatelná, a to díky své zjevné náhodné permutaci ukazatelů sufixů. Nedávné objevy v oblasti komprese dat tuto domněnku vyvrátily a otevřely dveře novým způsobům komprese sufixových polí. Tyto objevy jsou základním kamenem této datové struktury. Burrows a Wheeler navrhli transformaci sestávající z reverzibilní permutace, vytvářející text, který je jednodušší zkomprimovat. BWT má tendenci seskupovat znaky, které se vyskytují vedle podobných textových podřetězců. Tato vlastnost je poté využívána pomocí MTF kódování v kombinaci s některým statistickým kódováním (například Huffmano-

vými nebo aritmetickými kodéry), nebo strukturovanými kódovacími modely, k dosažení lepší komprimace.

Představme si, že máme vstupní řetězec $T[1, u]$, který transformujeme pomocí BWT na L, takže $L = BWT(T)$. Implicitní přítomnost sufixového pole A v L nám umožňuje plně využít strukturu A pro rychlé vyhledávání a velkou komprimaci L pro redukci prostoru. Malé prostorové nároky a rychlé vyhledávání jsou ultimátním cílem každého indexeru. [10]

4.3 Konstrukce a struktura FM-Indexu

Konstrukce samotné struktury se provádí během komprimace dat a celý FM-Index je pak tvořen z bloků. Každý blok pokrývá část zkomprimovaných dat a skládá se z pole s počtem výskytů jednotlivých symbolů a pro některé symboly i hodnotou, která je indexem označující pozici sufixu v původních datech.

FM-Index je možné použít při různých kombinacích kódování a pro libovolné komprimační algoritmy, je ale nutné aby jsme mohli blokově dekomprimovat data. Pro případy, že by byly použity algoritmy, které toto neumožňují, je možné do bloků uložit i další potřebná data, tak aby bylo možné blokově dekomprimovat.

4.4 Vyhledávání pomocí FM-Indexu

Vyhledávání pomocí FM-Indexu je prováděno nad sestavenou datovou strukturou a zkomprimovaným textem. Nechť $T[1, u]$ označuje libovolný text nad a nechť Z jsou zkomprimovaná data. V této části je popsán algoritmus, který při vzorku $P[1, p]$ hlásí všechny výskyty P v nekomprimovaném textu T při pohledu pouze na Z bez toho, aby jej musel celý dekomprimovat.

Nechť $T[1, u]$ označuje libovolný text, ve kterém chceme vyhledávat. V této části je popsán algoritmus, který při vzoru $P[1, p]$ hlásí všechny výskyty P v nekomprimovaném textu T při pohledu pouze na naši datovou strukturu a Z a bez nutnosti dekomprimovat celé Z. Náš algoritmus využívá vztahu mezi sufixovým polem a BWT transformací. Sufixové pole má dvě pěkné konstrukční vlastnosti, které jsou obvykle využívány pro podporu rychlého vyhledávání. Pro všechny sufixy textu T zabírá předpona vzorku P souvislou část. Souvislá část má počáteční pozici sp a koncovou pozici ep, kde sp je vlastně lexikografická pozice řetězce P mezi uspořádanou posloupností sufixů.

4.4.1 Vyhledání počtu výskytů vzorku P

Algoritmus 7 ukazuje, jak je prováděno vyhledání počtu výskytů vzorku P. [10] strana 393.

Algoritmus porovná vzorek od konce a využívá LF mapování. Funguje tak, že si zjistí index, na kterém začíná (sp) a končí (ep) v sufixovém poli poslední znak vzorku P a pomocí metody Occ zjistí počet výskytů předešlého znaku před tímto indexem.

Algorithm 7 Algoritmus pro spočítání počtu výskytů vzorku $P[1,p]$ v textu $T[1,u]$ pomocí datové struktury FM-Index

```
1: function FM_Count(P[1,p])
2:     c = P[p], i = p;
3:     sp = C[c], ep = C[c+1];
4:     while ((sp ≤ ep) and (i ≥ 2)) do
5:         c = P[i−1];
6:         sp = C[c] + Occ(c, 1, sp−1) + 1;
7:         ep = C[c] + Occ(c, 1, ep);
8:         i = i − 1;
9:     end while
10:    if ep < sp then
11:        return "pattern not found"
12:    else
13:        return "found (ep − sp + 1) occurences"
14:    end if
15: end function
```

4.4.2 Vyhledání indexů vzorku P v textu T

Vyhledávání indexů vzorku P je podstatně náročnější operace, protože musíme pomocí LF mapování nalézt index, pro který je uložen indexu v původním textu T. Provést pro všechny nálezy v rozmezí sp až ep.

5 Vlastní implementace FM-Indexu a experimenty

Kromě BWT transformace nevyžaduje FM-Index použití konkrétního algoritmu a je možné si vybrat libovolný. Pro jednoduchost byla proto zvolena následující kombinace: transformace Burrows-Wheller, následováno MTF kódováním a pro statickou komprimaci bylo použito Huffmanovo kódování.

5.1 Struktura bloků FM-Indexu

Na rozdíl od původního návrhu byly indexy pozic v originálním textu neukládány do bloků, ale uloženy bokem jako celek. Do bloků byly uloženy počty výskytů jednotlivých symbolů a nutné informace k umožnění blokové dekomprimace. Každým novým blok proto obsahuje stav MTF abecedy, bez níž by nebylo možné provést blokové MTF dekódování. Taktéž byla do každého bloku zapsána jeho bitová délku, díky níž je určováno kde začít dekódovat data pomocí Huffmanova kódování.

Implementace byla rozdělena do několila tříd. Jako první je FMIndex, kde se nachází aplikační logika pro vyhledávání. Tato třída využívá instanci třídy BucketInfo, která slouží jako správce všech bloků FM-Indexu. Stará se o vytváření bloků, naplnění správnými daty a o ukládání a načítání ze souborů. Vše lze vidět na třídním diagramu na Obrázku 4.

Třída Bucket reprezentuje samotné bloky a obsahuje všechny potřebná data pro vyhledávání a blokovou dekomprimaci. Třída BigBucket slouží jako pomocná třída a je používána pro reprezentaci všech bloků jako jednoho. Vlastnosti těchto dvou tříd lze vidět na třídním diagramu na Obrázku 5.

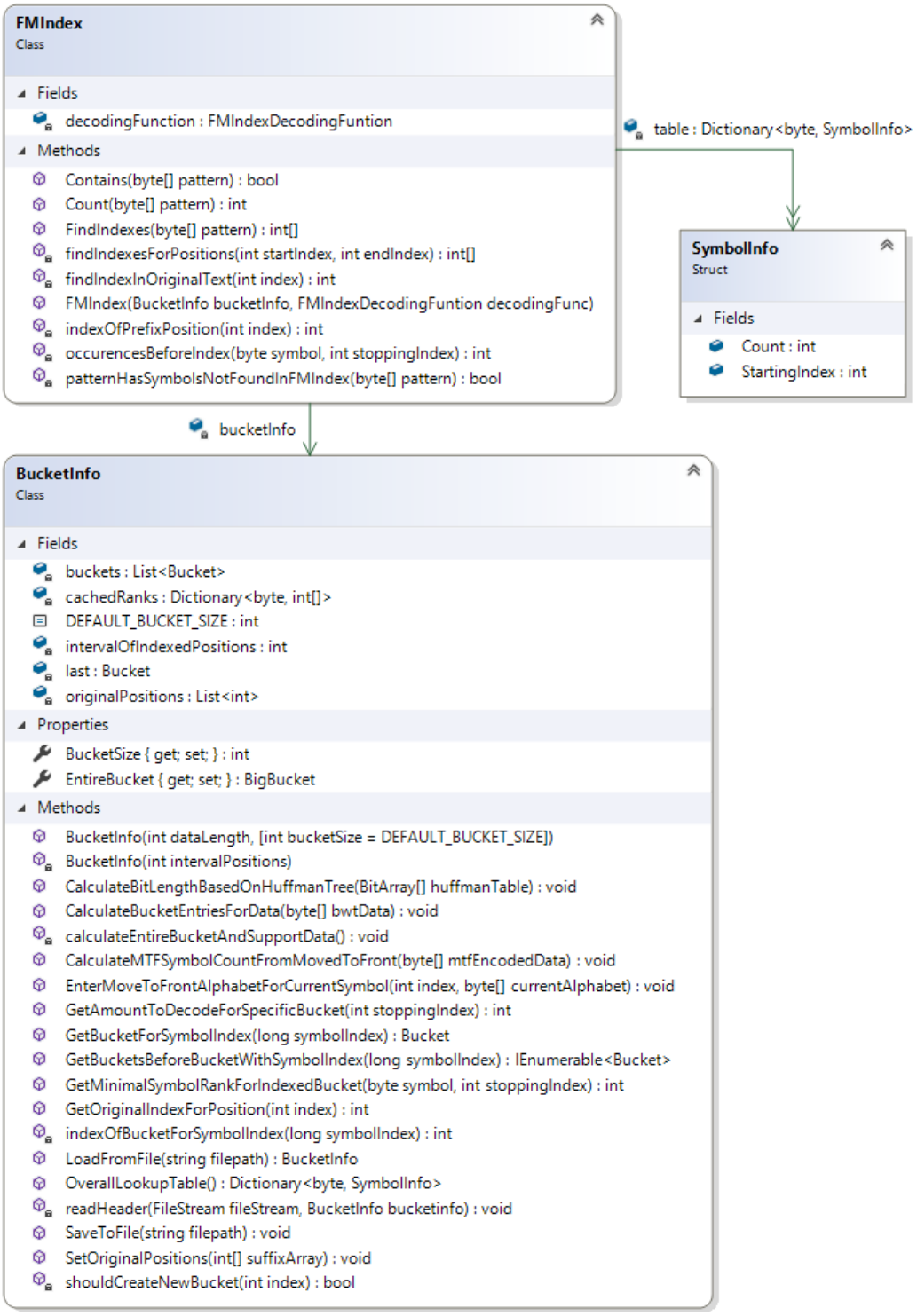
5.2 Vytváření FM-Indexu

Díky tomu, že lze FM-Index použít pro různé algoritmy, tak jsem se při implementování snažil, aby implementace nebyla nutně závislá na zvolené kombinaci komprimace. Tímto způsobem byla snaha dosáhnout toho, aby bylo možné vyzkoušet i jiné kompresní algoritmy, jiné kombinace kódování, jejich vliv na rychlost vyhledávání a nároky na prostor.

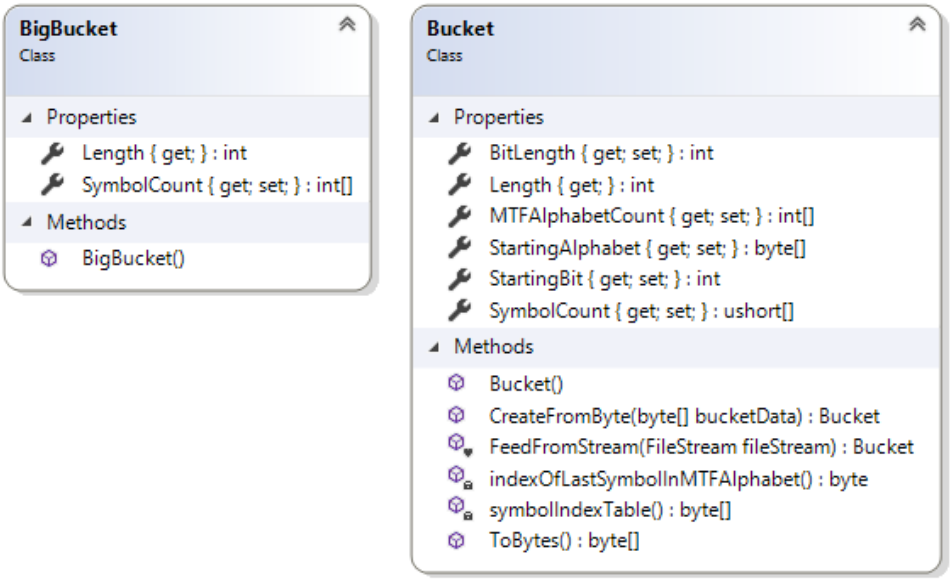
Proto byla snaha algoritmy pro komprimaci a samotné vytváření FM-Indexu rozdělit. Po provedení nějakého kroku komprimace, FM-Index analyzuje vzniklá data a zapíše si o nich informace, které potřebuje. Nevýhoda tohoto přístupu je rychlost vytváření FM-Indexu, v některých případech musí data znovu celé projít a proto může být konstrukce FM-Indexu pomalejší, nicméně pokud přihlédneme na celkovou dobu komprimace, tak je toto zpomalení zanedbatelné.

Jednotlivé kroky komprimace a vytváření FM-Indexu:

1. Provést BWT transformaci nad vstupními daty T, vznikne T_{bwt}.
2. Nad T_{bwt} provést MTF kódování. Během kódování se rozhoduje, kdy začne nový blok a pro každý nově začínající blok se zapíše stav MTF abecedy. Po MTF kódování vznikne T_{bwt_mtf}.



Obrázek 4: Třídní diagram FM-Indexu a BucketInfo



Obrázek 5: Třídní diagram bloků FM-Indexu, BigBucket a Bucket

3. FM-Index analyzuje T_{bwt} a do jednotlivých bloků zapíše počet výskytů symbolů, které pokrývají.
4. Následně analyzuje T_{bwt_mtf} a do bloků zapíše počet výskytů jednotlivých symbolů z MTF kódování, tento zápis provede do samostatné MTF tabulky.
5. Nad T_{bwt_mtf} je spuštěno Huffmanovo kódování a vznikne $T_{bwt\_mtf\_huff}$.
6. Z hlavičky Huffmanová kódování pro $T_{bwt\_mtf\_huff}$ zjistíme velikost kódů jednotlivých symbolů MTF tabulky a s její pomocí je vypočítána skutečná bitová délka jednotlivých bloků.

Z popisu komprimace a vytváření FM-Indexu logicky plyne, že kroky 2 a 3 by měli být obráceně. V této implementaci jsou tyto kroky prohozeny schválně a nejdříve je provedena MTF transformace. Během tohoto kódování je možné se lépe rozhodnout kdy započít nový blok. V případě vytváření bloků s proměnlivou velikostí by pak bylo možné vytvářet efektivnější bloky a tím celý proces vyhledávání urychlit.

5.3 Vyhledávání v FM-Indexu

Vyhledávání počtu výskytů vzorku lze vidět na Algoritmu 8. Algoritmus se moc neliší od původního návrhu, tak jak je vidět v Algoritmu 7, ale je napsán jinak, aby bylo jednodušší ho pochopit.

Než je spuštěna hlavní část algoritmu, tak je volána metoda $patternHasSymbolsNotFoundInFMIndex$, tato metoda slouží ke kontrole, zda hledaný vzorek neobsahuje znaky, které se v FM-Indexu nenacházejí. Logicky vyplívá, že by algoritmus takový vzorek nenašel. Pokud nám tato metoda

vrací *true*, tak je metoda okamžitě ukončena a vrací výsledek, že vzorek nebyl nalezen. Jedná o optimalizaci algoritmu tak, aby se zbytečně nerozbalovala data.

```
public int Count(byte[] pattern)
{
    if (patternHasSymbolsNotFoundInFMIndex(pattern))
        return 0;
    int i = pattern.Length - 1;
    byte currentSymbol = pattern[i];
    SymbolInfo currentSymbolInfo = table[currentSymbol];
    int startIndex = currentSymbolInfo.StartingIndex;
    int endIndex = startIndex + currentSymbolInfo.Count;
    while (startIndex < endIndex && i >= 1)
    {
        currentSymbol = pattern[i - 1];
        currentSymbolInfo = table[currentSymbol];
        startIndex = currentSymbolInfo.StartingIndex +
            occurencesBeforeIndex(currentSymbol, startIndex);
        endIndex = currentSymbolInfo.StartingIndex +
            occurencesBeforeIndex(currentSymbol, endIndex);
        i = i - 1;
    }
    // the interval is [startIndex, endIndex), pattern is at startIndex and
        before endIndex, not included at endIndex
    if (endIndex <= startIndex)
        return 0;
    return endIndex - startIndex;
}
```

Výpis 8: Vlastní implementace metody pro spočítání počtu výskytů vzorku

Nejvíce výpočtů se provádí v metodě $occurencesBeforeIndex$, která je popsána jako Algoritmus 9. Metoda vrací počet výskytů hledaného symbolu před daným indexem. Nejdříve je zjištěn počet výskytů symbolu v předešlých blocích a následně je prováděna kontrola, zda se tento symbol nachází v bloku, který kontrolujeme.

Pokud blok neobsahuje hledaný symbol, tak nemusí být rozbalen a je vrácen počet výskytů symbolu v předešlých blocích.

Pokud blok daný symbol obsahuje, tak je nutné jej rozbalit, aby byl zjištěn přesný počet výskytů symbolu před indexem. Rozbalen je pouze nutný počet symbolů a pro každý takto rozbalený symbol je provedeno porovnání, zda se jedná o ten hledaný. V případě shody je navýšen počet výskytů.

```
private int occurencesBeforeIndex(byte symbol, int stoppingIndex)
{
```

```
        int occurenceCount =
            bucketInfo.GetMinimalSymbolRankForIndexedBucket(symbol, stoppingIndex);
        Bucket bucketForCurrentIndex =
            bucketInfo.GetBucketForSymbolIndex(stoppingIndex);
        if (bucketForCurrentIndex.SymbolCount[symbol] > 0)
        {
            var startingBit = bucketForCurrentIndex.StartingBit;
            int numberToDecode =
                bucketInfo.GetAmountToDecodeForSpecificBucket(stoppingIndex);
            byte[] decodedBytes = decodingFunction(startingBit, numberToDecode,
                bucketForCurrentIndex);
            occurenceCount += decodedBytes.Count((decodedSymbol) => decodedSymbol
                == symbol);
        }
        return occurenceCount;
    }
```

Výpis 9: Metoda pro spočítání počtu výskytů symbolu před určitou pozicí

Po návratu počtu výskytů metoda *Count* zkontroluje *startIndex* a *endIndex*. Tyto dvě proměnné představují interval, ve kterém se může vzorek nacházet. Postupným procházením cyklu se tento interval zužuje, a to až do chvíle kdy zkontrolujeme všechny symboly vzorku P, anebo velikost intervalu bude rovna 0. Délka tohoto intervalu symbolizuje počet výskytů vzorku. Pokud se kdykoliv během vyhledávání dostaneme do situace, že $startIndex = endIndex$ tak se hledaný vzorek v textu T nenachází.

5.3.1 Vyhledávání indexů vzorku P

Vyhledávání indexů je podstatně náročnější operace pro FM-Index. Kvůli zajištění nízké spotřeby prostoru je uloženo omezené množství indexů originálních pozic. Dohledání indexu proto musí být realizováno pomocí LF mapování a to až do chvíle než narazíme na index, pro který je uchována hodnota indexu v původním textu.

Tento proces se musí provést pro každý nalezený index zvlášť. Kvůli své časové náročnosti proto nebudou experimenty využívající tuto operaci tak rozsáhlé.

Algoritmus nejdříve zjistí pozice *startIndex* a *endIndex*u stejně jako je tomu v metodě pro vyhledávání počtu výskytů. Po nalezení intervalu vyvolá metodu *findIndexesForPositions*. Tato metoda pro každý index v tomto intervalu zjišťuje jeho původní pozici. Algoritmus pro zjištění počtu výskytů lze vidět na Algoritmu 10.

```
    public int[] FindIndexes(byte[] pattern)
    {
        if (patternHasSymbolsNotFoundInFMIndex(pattern))
            return new int[0];
        var i = pattern.Length - 1;
```

```
        var currentSymbol = pattern[i];
        var currentSymbolInfo = table[currentSymbol];
        var startIndex = currentSymbolInfo.StartingIndex;
        var endIndex = startIndex + currentSymbolInfo.Count;
        while (startIndex < endIndex && i >= 1)
        {
            currentSymbol = pattern[i - 1];
            currentSymbolInfo = table[currentSymbol];
            startIndex = currentSymbolInfo.StartingIndex +
                occurencesBeforeIndex(currentSymbol, startIndex);
            endIndex = currentSymbolInfo.StartingIndex +
                occurencesBeforeIndex(currentSymbol, endIndex);
            i = i - 1;
        }
        // the interval is [startIndex, endIndex), pattern is at startIndex and
            before endIndex, not included at endIndex
        if (endIndex <= startIndex)
            return new int[0];
        return findIndexesForPositions(startIndex, endIndex);
    }

    private int[] findIndexesForPositions(int startIndex, int endIndex)
    {
        var indexesLength = endIndex - startIndex;
        var indexes = new int[indexesLength];
        for (var i = 0; i < indexesLength; i++)
        {
            indexes[i] = findIndexInOriginalText(startIndex + i);
        }
        return indexes;
    }
```

Výpis 10: Metoda pro zjištění všech indexů

5.3.2 Optimalizace algoritmu

Teoreticky by měl FM-Index s menšími bloky vyhledávat rychleji na úkor potřeby více prostoru. Z prvních experimentů bylo ale zjištěno, že pokud se FM-Index skládá z malých bloků, tak kolekce bloků narůstá do velkých rozměrů, což má negativní vliv při operaci pro výpočet počet výskytů. Jednoduchá operace jako je suma jednotlivých symbolů trvá příliš dlouho, pokud se musí provádět v řádu několika desítek tisíc součtů. Proto byl algoritmus optimalizován.

Jako řešení je při vytváření instance FM-Indexu předpřipravena tabulka se součtem výskytů symbolů v předešlých blocích. Tato tabulka vyžaduje další prostor a její velikost je přímo úměrná

počtu bloků a velikostí abecedy. Nicméně tato optimalizace zrychluje v některých případech vyhledávání až 100 násobně a hlavně umožňuje, aby nejsložitější operace algoritmu bylo rozbalování zkomprimovaných dat místo sčítání, díky tomu poté získáme časy vyhledávání reprezentující tuto strukturu.

5.4 Experimenty

V rámci experimentů byly porovnány rychlosti vyhledávání pomocí Boyer-Moore a datové struktury FM-Index. FM-Index byl testován při různých konfiguracích velikostí bloků a to v intervalu 500 až 50000.

Experimenty byly prováděny nad těmito typy dat:

- DNA - Skládá se z písmena A, C, G a T která reprezentují nukleotidy vláken DNA. Velikost symbolu je 2 bity, do jednoho bytu se tedy vejdou 4. Velikost abecedy je 16 symbolů.
- Anglický text - Texty knih v anglickém jazyce. Byly odstraněny hlavičky. Díky tomu, že se jedná o jazyk, tak se určité sekvence budou často opakovat. Velikost abecedy je 239 symbolů.
- XML - Skládá se popisků hlavních vědeckých článků. Velikost abecedy je 97 symbolů.
- Zdrojové kódy - Zdrojové kódy jazyka C a Java. Velikost abecedy je 230 symbolů
- Proteiny - Skládá se z protejnových sekvencí, každá nová sekvence začíná na novém řádku. Velikost abecedy je 27 symbolů.

Testovací data byly staženy z Pizza&Chili Corpus [12]. Z těchto testovacích dat byly náhodně vygenerovány vzorky různých velikostí, nad kterými byly následně prováděny experimenty.

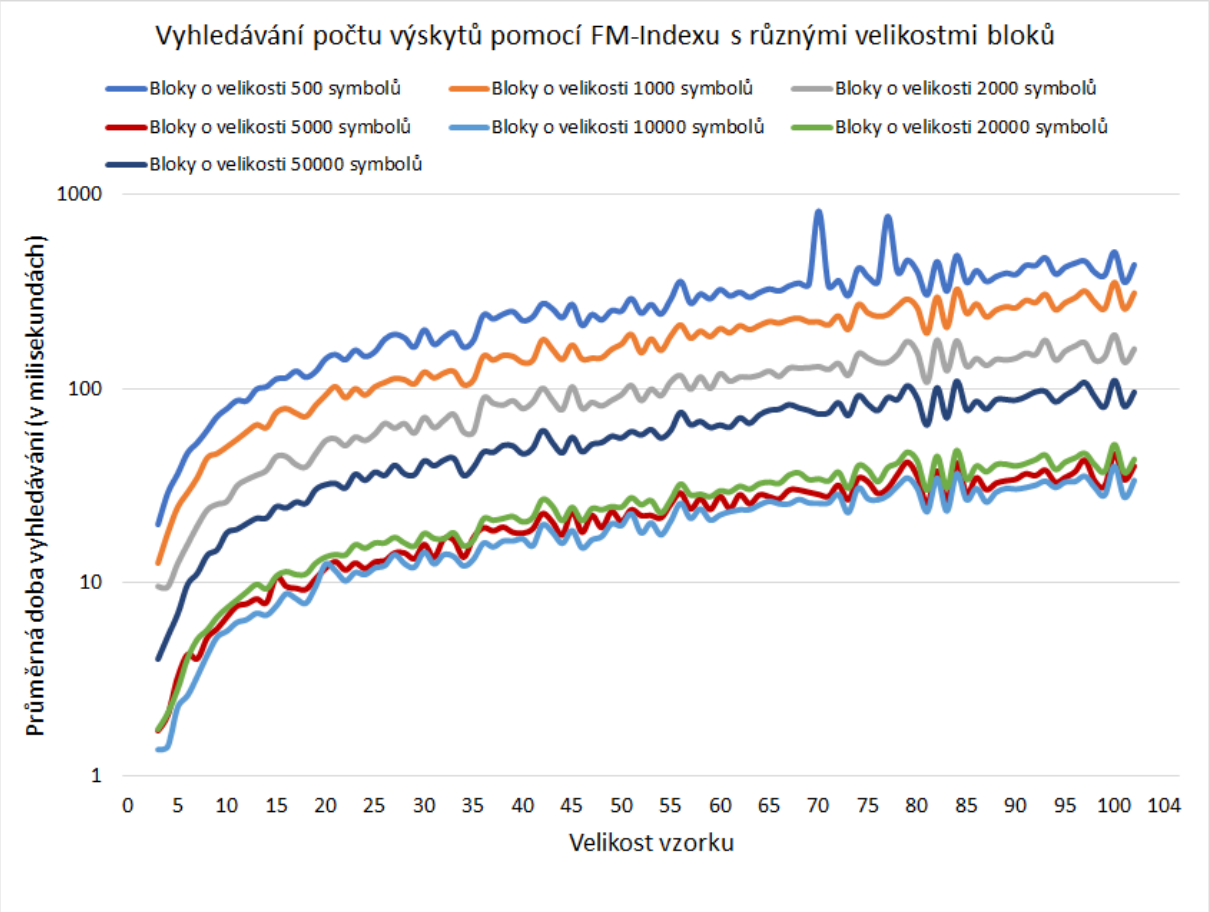
5.4.1 První experiment - anglický text

Tento test byl proveden nad anglickým textem o velikosti 100MB. Výsledky ilustruje graf na Obrázku 6. V grafu lze vidět, že s narůstající velikostí vzorku také roste čas vyhledávání, což je očekávaný stav.

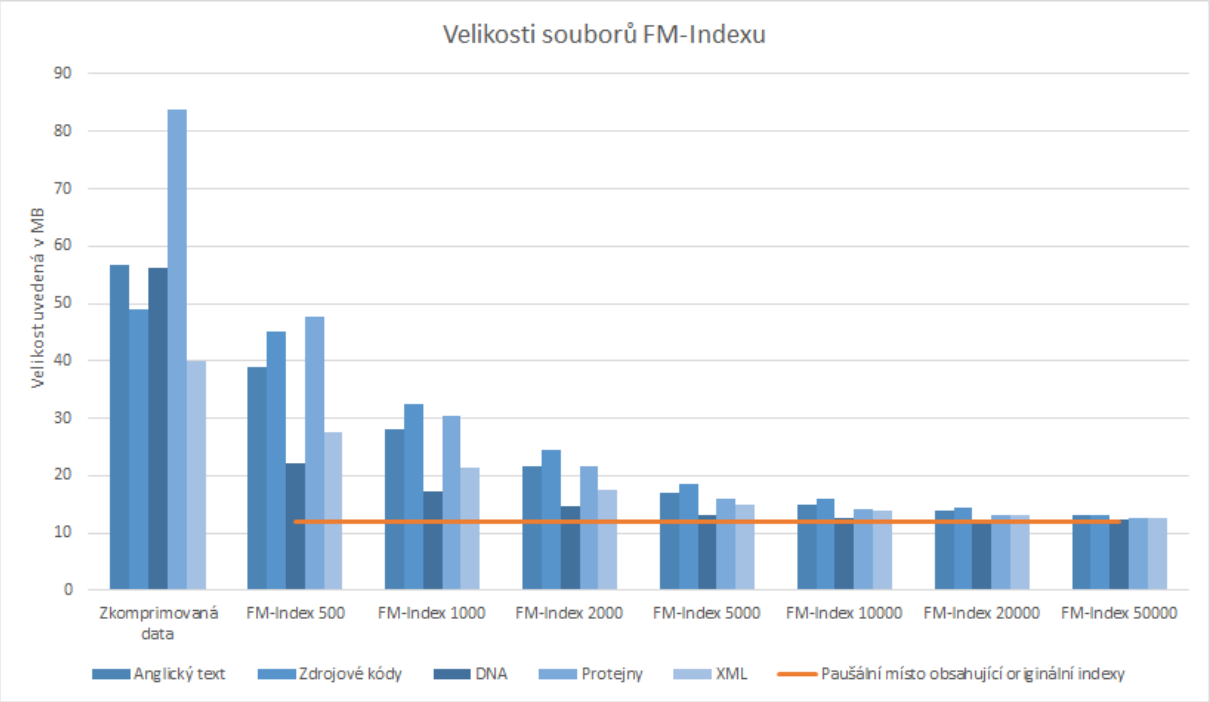
Nicméně, FM-Index s bloky o velikosti 500 symbolů je nejpomalejší a to i přesto, že by měl být nejrychlejší. Stejný problém se vyskytuje u FM-Indexu s velikostí bloků o 1000, 2000 a 5000 symbolech. Nejrychlejší je FM-Index s velikostí 10000, blízko následován FM-Indexem s velikostí 20000 symbolů na blok. Po těchto dvou je už s podstatným rozdílem FM-Index s velikostí 50000.

Z tohoto experimentu a blízkého pohledu na to jak algoritmus probíhal, bylo zjištěno, že FM-Indexy s malými velikostmi bloků jsou pomalé kvůli počítání součtu výskytů symbolů. FM-Indexy s velkými velikostmi bloků jsou pomalé kvůli velkému počtu dekomprimovaných symbolů.

Například FM-Index s bloky o velikostech 50000 symbolů musel dekomprimovat asi 100 krát více symbolů, než dekomprimoval FM-Index s velikostmi bloků 500 symbolů. I přesto byl



Obrázek 6: Vyhledávání vzorků pomocí FM-Indexu o různých velikostech



Obrázek 7: Velikost FM-Indexu podle velikosti bloků

FM-Index s bloky o velikostech 50000 symbolů podstatně rychlejší. Na základě výsledků tohoto experimentu byl algoritmus optimalizován, tak aby lépe odpovídal své podstatě a neztrácel výkon počítáním počtu výskytů v předešlých blocích.

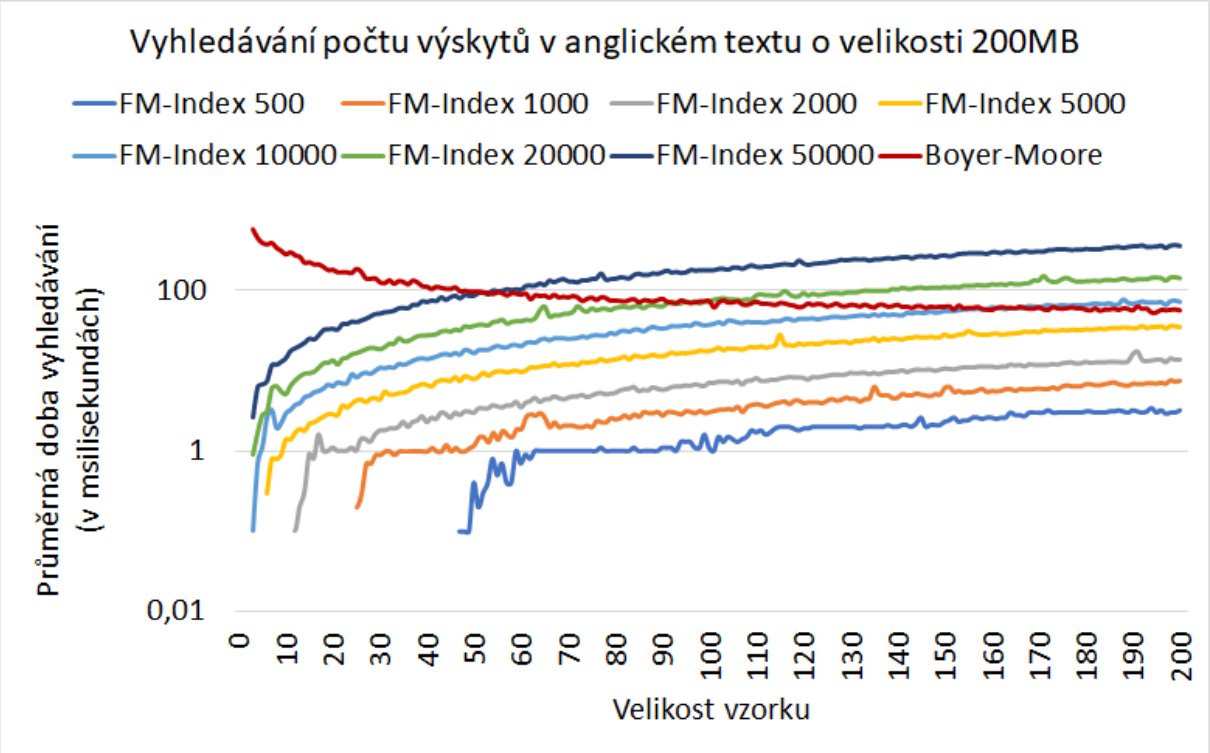
Druhý poznatek z tohoto experimentu je, že vzorky, které se v datech nenachází, mohou razantně zkreslovat výsledné statistiky. Toho si lze povšimnout v grafu u vzorků s velikostí v rozmezí 80 - 85. To samé pro vzorky o velikostech 68 a 76. Důvodem je to, jak funguje algoritmus, ten může v některých případech vracet okamžitě výsledek, že se vzorek v datech nenachází. Tyto výsledky velice zkreslují graf a proto nebudou v dalších experimentech používány vzorky, které se v datech nenacházejí.

5.4.2 Porovnání velikostí FM-Indexu

Nad vybranými testovacími daty byly vytvořeny datové struktury FM-Index. Vstupní soubor měl velikost 200 MB. Velikosti vytvořených souborů jsou ilustrovány v grafu na Obrázku 7.

V grafu je uvedena výsledná velikost zkomprimovaných dat, velikost FM-Indexu při různých velikostech bloků a znázorněna paušální velikost místo, ve kterém jsou umístěny originální indexy. Paušální místo, ve kterém jsou umístěny originální indexy je velké téměr 12MB a tato velikost je konstantní. Velikost bloků FM-Indexu nemá na toto místo žádný vliv a i kdyby se FM-Index skládal z jediného bloku, tak velikost celé datové struktury nemůže byt menší.

Z grafu si lze všimnout, že nejméně prostoru zabírají FM-Indexy vytvořenými nad DNA. Tento jev je dán tím, že většina bloků FM-Indexu vytvořených z DNA bude obsahovat stejný



Obrázek 8: Vyhledávání vzorků pomocí FM-Indexu o různých velikostech

symbol a nebude udržovat data pro další symboly. To samé se nedá říct o FM-Indexech vytvořených nad protejny, i přestože velikost abeced obou těchto formátu jsou podobné, tak kvůli způsobu uložení nemohou být protejny tak dobře zkomprimovány.

5.4.3 Testování vyhledání počtu výskytů

Nad vybranými testovacími daty byly spuštěny testy pro vyhledání počtu výskytů a porovnány vůči algoritmu Byoer-Moore. Výsledek pro anglický text lze vidět v grafu na Obrázku 8.

Z grafu lze vidět, že pro velmi krátké vzorky vyhledává Boyer-Moore velice dlouho a s FM-Indexem se nemůže srovnávat. Jakmile ale velikost vzorků dosáhne 50 znaků, tak nejpomalejší FM-Index, s velikostí bloků 50000 začíná ztrácet a z hlediska vyhledávání je pomalejší než Boyer-Moore. Na druhou stranu nejrychlejší FM-Index s velikostí 500 symbolů na blok teprve začíná vykazovat nějaké výsledky. Pro všechny vzorky menší než 50 symbolů byl FM-Index tak rychlý, že ani nebylo možné naměřit relevantní výsledky.

Z experimentů bylo zjištěno, že formát dat, ani jejich velikost nemají vliv na rychlost vyhledávání, alespoň ne pro vyhledávání počtu výskytů.

Pro vzorky, které se nenacházejí v datech se dá říci, že rychlost vyhledání je závislá, na jaké pozici ve vzorku dojde k tomuto zjištění. Například pokud bude zjištěno v polovině vyhledávání vzorku, že se daný v FM-Indexu nenachází, tak doba takového zjištění bude poloviny průměrné doby zjištění vzorku se stejnou velikostí, které se v datech nacházejí.

6 Závěr

Cílem této práce bylo naimplementovat datovou strukturu FM-Index a provést experimenty k porovnání vlastností této datové struktury vůči klasickým algoritmům pro vyhledávání vzorů. Stanovených cílů bylo dosaženo a z výsledků experimentů byla zjištěna rychlost vyhledávání a prostorové nároky při různých konfiguracích.

V práci byly uvedeny a vysvětleny jakým způsobem pracují algoritmy pro vyhledávání v nekomprimovaných datech a navazující indexovací datových struktury. Potíže spojené s používáním datových struktur, leč byly rychlejší než klasické algoritmy a jejich postupným vyřešením, který vedl až k jejich samotné komprimaci a dal vzniku datové struktuře FM-Index.

Tato datová struktura byla podrobněji vysvětlena a popsána v teoretické i praktické části této práce. Nad vlastní implementací byly provedeny experimenty, z nichž byly vyvozeny následující závěry.

Podle očekávání bylo prokázáno, že datová struktura FM-Indek je rychlejší na vyhledávání než klasické algoritmy. Bylo zjištěno, že čím menší je blok FM-Indexu, tím méně je potřeba dekomprimovat data a díky tomu je vyhledávání rychlejší. Zrychlení vyhledávání oproti konfiguracím, kdy je blok FM-Indexu n-krát větší je n, to znamená, že FM-Index s 10krát menší velikostí bloků vyhledává 10krát rychleji, než FM-Index s bloky takové velikosti. Velikost vstupních dat, anebo jejich formát dat neměly na rychlost vyhledávání žádný vliv a rychlost byla závislá na velikosti bloků FM-Indexu a na velikosti vyhledávaného vzorku. Formát dat měl vliv na finální velikost datové struktury FM-Index a na velikost zkomprimovaných dat.

Literatura

[1] Maxime Crochemore, Christophe Hancart, Thierry Lecroq: Algorithms on Strings (2007)

[2] Knuth, Donald; Morris, James H.; Pratt, Vaughan: Fast pattern matching in strings (1977)

[3] Maxime Crochemore, Thierry Lecroq: Pattern matching and text compression algorithms (1996)

[4] Amir A., Benson G., Farach M.: Optimal two-dimensional compressed matching (1994)

[5] Barna Saha: Pattern Matching in Compressed DNA Sequence (2008)

[6] Lei Chen, Shiyong Lu, Jeffrey Ram: Compressed Pattern Matching in DNA Sequences (2004)

[7] R.S. Boyer, J.S. Moore: A Fast String Searching Algorithm. Communications of the ACM, 20, 10, 762-772 (1977)

[8] Donald Adjeroh, Tim Bell, Amar Mukherjee: THE BURROWS-WHEELER TRANSFORM: Data Compression, Suffix Arrays, and Pattern Matching (2008)

[9] M. Burrows and D.J. Wheeler: A Block-sorting Lossless Data Compression Algorithm, (1994)

[10] P. Ferragina ; G. Manzini: Opportunistic data structures with applications, (2000)

[11] D. Salomon: Data Compression: The Complete Reference 4th Edition, (2006)

[12] http://pizzachili.dcc.uchile.cl/texts.html

A Elektronické Přílohy

Obsah přílohy:

- Složka Sources - Obsahuje zdrojové kódy aplikace
- Složka Dist - Obsahuje konzolovou aplikaci řešící problematiku práce
- Uživatelská příručka pro aplikaci
- Složka Test - Obsahuje testovací data